WHITEPAPER

Endian: From the Ground Up

A Coordinated Approach

By Kevin Johnston
Senior Staff Engineer, Verilab

July 2008

7320 N MOPAC Expressway | Suite 203 | Austin, TX 78731-2309 | 512.372.8367 | www.verilab.com

Data in computers regularly undergoes dozens if not hundreds of transformations between producer and consumer.

INTRODUCTION

Imagine XYZ Corp finally receives first silicon for the main chip for its new camera phone. All initial testing proceeds flawlessly until they try an image capture. The display is completely garbled.

Of course there are many possible causes, and the debug team analyzes code traces, packet traces, memory dumps. There is no problem with the code. There is no problem with data transport. The problem is eventually tracked down to the data format.

The development team ran many, many pre-silicon simulations of the system to check datapath integrity, bandwidth, error correction. The verification effort checked that all the data submitted at the camera port eventually emerged intact at the display port; but the data was just numbers. It wasn't until the data was rendered as an image that it became obvious that Something Was Wrong.

The image grabber produced "little endian" data, the processor is "big endian", and the error so painfully obvious in the image display is an example of the "endian problem".

This may sound like a contrived and implausible story. Contrived, yes; implausible, sadly no. Data in computers regularly undergoes dozens if not hundreds of transformations between producer and consumer. From parallel to serial and back, serial code conversion (e.g. 8B/10B), it gets broken into network packets and reassembled, there are audio encoding standards, video encoding standards, different encodings for still video vs motion video, there are color code conversions, compression and decompression, and on, and on, and on. Some conversions are performed in hardware, some in software, some by a combination of both.

With all this going on, it's actually perilously easy to lose sight of what the numbers all mean, and the endian problem is all about the most fundamental meaning of a number: What value does it represent?

This document explores endianness using a coordinate system terminology, and explains common endian misconceptions as arising from coordinate system ambiguity. Topics covered are: The definition and properties of endian; the endian problem; independence of bit and byte endian; why bit significance and byte address must be managed in hardware; why byte significance must be managed in software.

WHAT DOES ENDIAN MEAN?

Endian relates the **significance order** of symbols to the **position order** of symbols in any representation of any kind of data, **if** significance is position-dependent in that representation.

Let's take a specific type of data, and a specific form of representation that possesses position-dependent significance: A digit sequence representing a numeric value, like "5896". Each digit position has significance relative to all other digit positions.

I'm using the word "digit" in the generalized sense of an arbitrary radix, not necessarily decimal. Decimal and a few other specific radixes happen to be particularly useful for illustration simply due to their familiarity, but all of the discussion still holds in base 17 or any other except infinity and 1.

Using the digit sequence as a point of reference, we can state certain universal properties of endianness:

- *P1:* Endianness is a property of a **symbolic representation** of a numeric value. The number represented does not possess endianness; it is the representation of the numeric quantity as a sequence of digit symbols that possesses endianness.
- *P2:* Endianness is a property of a representation of a **single** numeric quantity. It's not about relative magnitude of multiple numeric quantities, as in sorting independent numbers into any kind of order based on magnitude.
- *P3:* Endianness is a **private, local property** of any representation of any number. Every unique representation of any numeric quantity has its own endianness in any given digit position coordinate system, independent of any other representation of the same or any other numeric quantity in the same or any other digit position coordinate system. The endianness of a number representation depends on the position and significance of its digits, and nothing else.
- *P4:* Endianness does not exist in an infinite radix notation. In an infinite radix, any numeric value is expressed by a single symbol, and a single symbol does not possess endianness: Endianness requires multiple digits with position dependent significance. Corollary: In **any** radix, values less than the radix can be expressed as a single digit, and therefore do not possess endianness.

- *P5:* Endianness does not exist in base 1 notation. Base 1 digits do not possess position dependent significance — a number is represented by the count of "marks", so e.g. if we use the vertical bar symbol as the mark, then the number 3 could be represented ||| and the number 7 could be represented |||||||. No mark is any more or less significant than any other.

Base 1 is actually in fairly widespread use: Think of dice, dominoes, playing cards. Dice do not possess endianness.

Numeric digits don't always possess positional significance: in "1, 2, 3, go!" the digits 1, 2, 3 don't have positional significance. They don't represent the single value one hundred twenty-three, they represent independent values one, two, and three — and would still represent those values in any order. The '3' wouldn't change significance to mean three hundred if you said "3, 2, 1, go!" instead.

On the other hand, if the digit sequence 123 is used to represent the numeric value one hundred twenty-three, then the digits do possess positional significance, and therefore endianness. But it's still not possible to say what endianness they possess. We haven't yet defined a frame of reference to relate the digit positions. This is where the notion of a coordinate system enters the picture. A left-right axis might be usable to identify the digit positions, but it's not the only option. A North-South axis may not be useful at all.

To ascertain the endianness of a number representation, you must know two things: The relative significance of digits and the position coordinate system. Often, but not always, the position coordinate system degenerates to simple **ordinality** (digit position 0, digit position 1, digit position 2, and so on). The dependence on digit position gives rise to the great granddaddy of all endian properties:

- *P6:* **A single representation of a single numeric value can simultaneously possess different endiannesses according to different digit position coordinate systems.** The number representation doesn't change, the number represented doesn't change, but if you adopt a different coordinate system, the endian can change. Much of the confusion surrounding endian stems from ambiguity of the digit position coordinate system in which a given endian is stated — and can be resolved by more diligent attention to the coordinate system.

Given any two of these three pieces of information — the relative significance of digits, the coordinate system, and the endianness — you can derive the third. In order to interpret a digit sequence as a numeric value, the only fact you really need is the significance of the digits. If you happen to know that directly, there's no need to be concerned with either of the others; but if you don't know it directly, then you need both of the others to derive it.

BUT WHAT DOES ENDIAN ***REALLY*** MEAN?

According to Wikipedia:

> In computing, **endianness** is the byte (and sometimes bit) ordering in memory used to represent some kind of data. Typical cases are the order in which integer values are stored as bytes in computer memory (relative to a given memory addressing scheme) and the transmission order over a network or other medium. When specifically talking about bytes, endianness is also referred to simply as **byte order.**

and:

> Integers are usually stored as sequences of bytes, so that the encoded value can be obtained by simple concatenation. The two most common of them are:
>
> - increasing numeric significance with increasing memory addresses, known as *little-endian*, and
> - its opposite, most-significant byte first, called *big-endian*.

"increasing numeric significance with increasing memory address" is a relation between significance order and position order, in the memory address coordinate system. If a unique address identifies a unique byte of memory, then each byte is a digit, and a concatenation of bytes with position dependent significance is a base 256 representation of numeric value. (Yes, each entire byte is a single digit; the address coordinate system doesn't resolve position finer than a byte granularity.)

So far, so good. But beware of any endian definition that uses the word "first". Watch carefully:

> "the Hindu-Arabic numeral system is used worldwide and is such that the most significant digits are always written to the left of the less significant ones" (S1)

A single representation of a single numeric value can simultaneously possess different endiannesses according to different digit position coordinate systems.

So for the record, the definition of endian in the left-right coordinate system is: Most significant digit on the left is big endian; most significant digit on the right is little endian.

Most significant digit on the left. Always. But then:

> "Writing left to right, this system is therefore big-endian (big end first). Writing right to left, this numeral system is little-endian" (S2)

Ever seen (or heard) this statement before? Do you agree? I think it's misleading. Quickly: What's the coordinate system?

Instead of "big endian means big end first", try this definition: "Big endian means big end (most significant digit) nearest the origin". They both mean the same thing if you know the coordinate system, but "origin" pretty much forces you to think in terms of coordinate system, and "first" doesn't. If you think in terms of a coordinate system, you cannot go wrong.

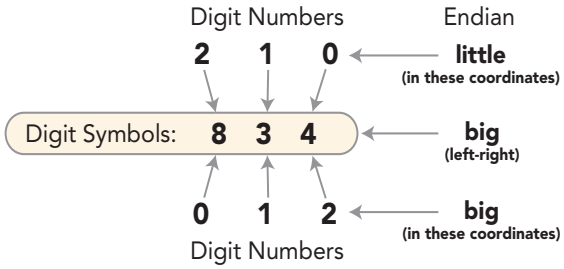
If you are writing left-to-right, where's the origin? There's not enough information to tell! The direction of writing doesn't determine the coordinate system. In fact, "direction" is not even defined unless you first choose a coordinate system. So for the record, the definition of endian in the left-right coordinate system is: Most significant digit on the left is big endian; most significant digit on the right is little endian. Writing direction makes no difference whatsoever. Endianness is a local property of an individual number representation; it depends on the significance and position of its own digits, and nothing else (P3). If Hindu-Arabic numerals are always written with the most significant digit on the left (S1), then they are always big endian in the left-right coordinate system. End of story.

But if you change coordinate systems, the endianness can change too (P6). If you write right-to-left, it may seem completely natural to adopt a coordinate system with the origin on the right, and voila: you write Hindu-Arabic numerals little endian. But it's the change of coordinate system that made the difference, not the direction of writing. In a **right-origin coordinate system**, Hindu-Arabic numerals are **always** little endian. The writing direction (still) makes no difference whatsoever. And regardless, left-right endian is not defined in a right-origin coordinate system. Left-right endian is defined in a left-origin coordinate system. It's an arbitrary choice, but that is the definition.

Statement S2 is not true in either a left-origin or a right-origin coordinate system. But it is true in the chronological coordinate system. If earliest time is the origin, and you write the most significant digit first (chronologically), that's chronological big endian. If you write the least significant digit earliest, that's chronological little endian.

The reason I don't like S2 is that I don't consider chronological order to be an intuitive or useful coordinate system for written numbers. Chronological endian is relevant to spoken numbers, and critical for computer data communication, but irrelevant for written numbers. Even worse, if you thought S2 was talking about a left-right coordinate system, you could easily form some very wrong notions of endianness, like "endian depends on the writing direction of the surrounding language". And then you'd be pretty stuck if there were no surrounding language text for reference, or if you don't recognize the language and you don't know in which direction it's written.

In addition to the left-right and chronological axes, we can use ordinal enumeration to identify digits. If we take the digit sequence 834 to represent the numeric value eight hundred thirty four, and we enumerate the digit positions, that's a coordinate system and it establishes an endian. But the endian depends on the enumeration:



Enumerating the digit positions by their radix exponent always produces a little endian enumeration. Increasing significance with increasing position is little endian — "little end first". This doesn't make this coordinate system any more or less valid than any other.

I read somewhere that all the earliest mainframe computers were big endian, and that little endian was only introduced with later mini- and micro-computers. I suspect this was because storing values in memory in little endian byte address order was beneficial for coding multi-precision arithmetic in software: For addition and subtraction, you first align the operands on the radix point — the "little end" — and then you start at the "little end" and propagate carry/borrow on to the more significant digit computations. This might also explain why little endian machines store text strings in big endian byte address order (most significant character at the lowest address). Text strings can't be meaningfully added or subtracted, but they can be compared, and for string comparison, the radix alignment point is the most significant character — the "big end". No endian storage order works perfectly for all cases. For a numeric compare operation, the values still have to be aligned on the radix point — the "little end" — but even then, when comparing numbers, it's still more efficient to start at the "big end".

I've heard people use S2 to explain the endian problem. Essentially, the argument is that you need to know the language direction to know the significance of the digits. This is false: you don't need to know the language direction to know the significance of the digits. In everyday experience, it is always possible to interpret a digit sequence in the Hindu-Arabic form, because this form is always big endian in left-right coordinates — most significant digit on the left (S1). If you don't design computers or write programs, it's entirely possible you could go through your entire life without ever worrying about endianness. The language direction analogy utterly fails to capture the essence of the endian problem because there is no endian problem in written Hindu-Arabic numerals.

One of the very few examples of the endian problem in common experience is the date problem: In England, dates are commonly written dd-mm-yy (or yyyy); in the US, dates are commonly written mm-dd-yy (or yyyy); and in ISO-8601, dates are written yy-mm-dd (or yyyy). How can you tell whether 04-01-06 means January 4th 2006 or April 1st 2006 or January 6th 2004? Without any kind of contextual cue, the answer is: "You can't". It is genuinely impossible. **This** is the endian problem: You must know the format to understand the value.

In computers, the endian problem is compounded:

1. Bits and bytes have independent ordinalities, and neither bits nor bytes are universally one endian or the other. (I'm using the word "ordinality" to emphasize that "left-right" is rarely a useful coordinate system for bits and bytes in situ in a computer. The coordinate system of both bits and bytes is usually ordinal enumeration, but not the same ordinals.) For an analogy, the yy, mm, dd date fields do not adhere to any single endian standard: yy-mm-dd is big endian, and dd-mm-yy is little endian (left-right coordinate system), and both are in common use. But the digit pairs in each field do adhere to a single endian standard: The Hindu-Arabic form. "01" never means "October". In computers, though, both bits and bytes come in both endians.

2. Bit endian can change at different locations in a system, because the presiding coordinate system can change.

3. Written languages use separate symbol sets for representing numeric data (a digit symbol set), text data (an alphabet symbol set), and other syntactic functions (space, punctuation symbols). Computers use the same symbol set (bits) for all these and more, and the only thing that distinguishes any bit from any other is context.

If it were truly an insurmountable problem then only one standard computer endianness would have survived, analogous to the one standard endianness for written Hindu-Arabic numbers. For better or worse, it is not insurmountable, and both endians have survived. And the solution is context; but **context does not determine endianness:**

1. Context is merely a hint to guide the interpretation of data; it tells a data consumer what endianness to expect. It is up to the data producer (some program) to produce data with the appropriate endianness, and it is up to the data consumer (some other program) to operate on the data appropriately. Context is passive; it's a contract, not a broker. (For the purposes of this discussion, a coordinate system is **not** a "context". They are not interchangeable terms. Position (coordinate system) can exist without significance (data), and vice versa. Neither the coordinate system nor the data individually know anything about endianness. Endianness is about how data is embedded in a coordinate system. Context is neither the coordinate system nor the data, it's metadata about the embedding; and more precisely, it's a statement of presumed embedding, not necessarily the actual embedding.)

2. Changing context does not change data, nor does it change the coordinate system. Imposing a new context on a region of memory doesn't change the memory contents or the memory addresses; any number representations present remain where they were, with the same endian they had before. If they don't conform to the new context, then they don't conform; and will almost certainly be misinterpreted by any consumer expecting the new context format. In the new context, what was originally a number representation may occupy space that is no longer intended to contain a number representation at all. It still is a number representation, it still has the same endian, but it won't have the same meaning (its original numeric value) to a consumer expecting something different.

Endian is a fact: a property possessed by a representation of numeric value with position dependent digit significance.

Neither the coordinate system nor the data individually know anything about endianness. Endianness is about how data is embedded in a coordinate system.

The bit numbers are completely independent of the byte addresses; they're independent coordinate systems.

Context establishes an expectation of format to be assumed by something operating on data. If the expectation matches the fact, then the data is intelligible; otherwise, it's meaningless. You can be told that a date is in British format; maybe it's written on a British customs form that says "Arrival and Departure dates (dd/mm/yy)". But if the form was filled out by a tired American in the poor light of a cramped seat of a loud airliner, the contextual expectation of format is not a guarantee.

BIT SIGNIFICANCE

In memory, it is almost universal for unique addresses to identify individual bytes of data. Logically, each byte is composed of 8 discrete bits, but the individual bits are not directly addressable. Address is an ordinal enumeration: a coordinate system with the origin at the lowest address.

Even though they are not individually accessible by address, the individual bits composing each byte in memory may have an ordinal enumeration of their own. These bit numbers are used in diagrams or a hardware description language of the memory circuit design. This ordinal bit enumeration is a coordinate system with the origin at the lowest bit number. The bit numbers are **completely independent** of the byte addresses; they're independent coordinate systems. This is partly because they're not even enumerating the same structures — bits vs bytes — but more fundamentally because all possible coordinate systems are inherently independent. A coordinate system "just is." Very few people other than memory circuit designers ever use, or even know, the memory circuit bit enumeration. Most anyone else will probably use bus bit numbers.

If multiple wires (a "bus") are employed to transfer data in parallel (multiple bits at the same time, one per wire) between memory, processor, and other components of a system, each individual wire is typically identified with a unique wire number — often referred to as a bit number. These bit numbers are another ordinal enumeration: a coordinate system with the origin at the lowest bit number. The bus wire/bit numbers are **completely independent** of memory circuit bit numbers and memory byte addresses. Even when enumerating the same structures (bits), coordinate systems are independent.

Inside a data processing unit, it is almost universal to identify each individual bit of data with a unique number. These processor bit numbers are **completely independent** of bus bit numbers, memory circuit bit numbers, and memory byte addresses.

Here are some byte values, expressed in both hex (base 16) and binary, in a few specific memory addresses:

Address	0	1	2	3
Byte value (H)	01	23	45	67
Byte value (B)	00000001	00100011	01000101	01100111

Even though you know the positions of these bytes, you don't know their endian in the address coordinate system, because you don't know their relative significance. They're just byte values in memory. And they may not even have relative significance: They could be individual byte representations of independent numeric values.

If I tell you that the bit representations of the byte values are left-right big endian, then you know the significance of each bit, but you don't know the bit endian in the bit number coordinate system, because I didn't give you the bit numbers.

What if I told you that the binary representation is actually left-right little endian? Well, it's not, and it couldn't be: 0111 is not the left-right little endian binary representation of hex 7. The hex equivalent of left-right little endian binary 0111 is E. Even though "E" and "7" are single digits, and don't possess endian themselves (P4), they tell you the significance of the digits in the binary form. Define a binary digit position coordinate system, and you get a bit endian. The hex digits don't tell you the endian of the binary digits until you define a binary digit coordinate system. In fact, the hex digits don't have a known endian either, until you define a hex digit coordinate system. Byte address is not a hex or a binary digit coordinate system.

If the binary representations were really truly left-right little endian, would the hex equivalent of byte address 3 be 6E or E6? It's a trick question: It could be either. Left-right little endian binary 01100111 is equivalent to left-right little endian hex 6E; but it's also equivalent to left-right big endian binary 11100110 and left-right big endian hex E6. They all represent the same left-right big endian decimal value 230. But notice that changing the endian (from big to little) while keeping the same coordinate system (left-right) changed the significance of the digits. Left-right big endian binary 01100111 is equivalent to left-right big endian hex 67 is equivalent to left-right big endian decimal 103.

Other system components attached to the same bus may internally number the individual bits presented to or obtained from the bus. Each such numbering scheme is **completely independent** of all others, of processor bit numbers, of bus bit numbers, of memory circuit bit numbers, and of memory byte addresses.

Notice that there are possibly many coordinate systems for bits, but only one coordinate system for bytes. There is a reason for this, discussed in "Byte Address" below.

In all computer data communication, it is almost universal that the smallest quantity of data transportable from one location to another is a byte, because bytes are the smallest addressable data units: atomic radix-256 "symbols". However, internally a byte is composed of 8 bits with position-dependent significance: **bit endianness**. Hopefully it's not too surprising that "increasing significance with increasing bit number" (least significant bit at the origin; "little end first") is called bit little endian, and "decreasing significance with increasing bit number" (most significant bit at the origin; "big end first") is called bit big endian. Take your pick of coordinate system; you get an endian in that coordinate system.

For every single component in a system, even the wires that conduct the signals from here to there, the designer can adopt whatever bit coordinate system he or she pleases, but byte symbols have to mean the same symbol to all — because they're the atomic symbols of communication. Therefore, bit significance within byte atoms must be known, and must be preserved. So if you want to join two components of known opposite bit endianness, you must reverse the bit coordinate system across the interface. Increasing bit numbers on one side must mate with decreasing bit numbers on the other side.

WARNING: This rule applies only **within** bytes. A different rule applies to separate bytes. More below.

But what if you don't know the bit endian of some piece of a system (device, bus, anything)? Well, it's impossible not to know bit endian: Bit number is a coordinate system, and bit significance must be preserved, and between them they define a bit endian.

Back to Wikipedia again:

> Most modern computer processors agree on bit ordering "inside" individual bytes (this was not always the case). This means that any single-byte value will be read the same on almost any computer one may send it to...

The second sentence is indisputably true, but I think the first sentence is questionable. What does "bit ordering" mean? My best guess is that it is claiming that most modern processors use the same bit number for the same significance position within bytes. And that's not true; it never was true in the past, and it's not true today. x86 processors use little endian bit numbering, PowerPC processors use big endian bit numbering, others are split on the issue. What is true today is that all modern systems agree to maintain bit significance (within individual bytes) across all interfaces, and sacrifice bit number correspondence whenever and wherever necessary.

Single-byte values don't automatically work because of any bit numbering agreement; they don't automatically work just because it's somehow no longer an issue in "modern" computers. Bytes work because computer hardware designers carefully and deliberately Make It So. It is not an issue for programmers or end users, because it's managed entirely within the hardware; but it most assuredly is an issue to the hardware designer who gets it wrong.

Since all bit enumerations are completely independent of byte addresses, **bit endian is completely independent of byte endian.** There is no relationship whatsoever. Furthermore, byte address is completely independent of byte significance, so address itself is also completely independent of byte endian.

BYTE ADDRESS

Bit significance within bytes is conserved by *bit swapping* across any bit-endian-mismatched interface. (It's really the bit coordinate system that gets swapped, and endian along with it. The bit values, with their significance, propagate straight through the connection. Bit significance is not swapped!)

Don't try this with bytes.

Can't we likewise preserve byte significance by *byte swapping* across a byte endian mismatched interface?

No. It doesn't work. Actually, it's far worse: it may work sometimes, but not other times.

Bit swapping in bytes works because all bytes are exactly the same size: A byte is 8 bits. Byte swapping within larger data structures does **not** work because all data structures (variables) are **not** the same size. (It also doesn't work if variables aren't size-aligned, but that can usually be avoided, and should be avoided as much as possible. Size misalignment can cause very significant performance degradation even when endianness is not an issue.)

For every single component in a system, even the wires that conduct the signals from here to there, the designer can adopt whatever bit coordinate system he or she pleases...

Why does data size make a difference? A byte is an atomic symbol, and all 8 bits contribute to the numeric value: significance extends across all bits. Since bytes are atomic, the boundaries between different variables always occur between bytes. If two adjacent (consecutive address) bytes are not parts of the same variable, then significance does not extend across that byte boundary, and it is wrong to swap bytes across that boundary. You'd be jumbling pieces of different variables together.

But wait. Isn't it just as wrong not to swap bytes that are part of the same variable? I know it's tempting, but it's extraordinarily rare that an interface:

1. Knows where the variable boundaries are at all;
2. And furthermore knows the byte endian of every variable;
3. And even knows the desired byte endian where the data is going.

No widely used bus standards even have wires to supply this information.

Because of different data sizes, any byte boundary of an interface could be within a single variable or between two different variables at any time.

It's wrong to swap bytes across variable boundaries, and it's just as wrong to swap a 4-byte variable as two 2-byte halves. Because of different data sizes, any byte boundary of an interface could be within a single variable or between two different variables at any time. **Don't byte swap across interfaces.**

Consider a few variable declarations (assume the "quadlet" data type is four bytes, the "doublet" data type is two bytes, and the "byte" data type is one byte):

```
quadlet alpha;
doublet bravo;
byte charlie;
byte delta;
```

The compiler will allocate memory locations for each variable, and throughout the rest of the compiled program, each variable is referenced by memory address. By arbitrary convention, "the address" of a variable is the lowest byte address of all its constituent bytes. To simply allocate memory space for variables, the compiler need neither know nor care about byte endianness. You can compile and run the exact same program source on a big endian machine and a little endian machine. Assuming the compiler doesn't generate any byte swapping instructions on either machine, the big endian machine will store the most significant byte of variable alpha in the lowest byte address of the allocated space, and the little endian machine will store the least significant byte of alpha in the lowest byte address of the allocated space. If a computer system includes both a big endian processor and a little endian processor, each executing an independent instance of the program (compiled to its own instruction set, with no byte swapping), with independent memory space allocated for its variables, both processes can happily run side by side with no interaction, and with no problem.

However, if you share memory between the two processors, life can quickly become much less happy. There is no data width that you can simply swap bytes at any interface that will allow both processors to work in the same memory space. To operate correctly, `alpha` would have to be byte swapped as a four byte structure (swapping the lowest address byte with the highest address byte, and also swapping the two middle address bytes), `bravo` would have to be byte swapped as a two byte structure, and `charlie` and `delta` should not be swapped at all:

Assume `alpha=0x01234567` (decimal 19,088,734), `bravo=0x89ab` (decimal 35,243), `charlie=0xcd` (decimal 205), `delta=0xef` (decimal 239) (all values expressed left-right big endian).

Variables stored in memory in big endian byte address order (hex representation of byte value is still left-right big endian):

Variable:	alpha	bravo	charlie	delta
Address:	0 1 2 3	4 5	6	7
Byte value:	01 23 45 67	89 ab	cd	ef

Values (left-right big endian) of variables that each processor would interpret if reading from memory where the variables are stored in big endian byte address order:

	Big endian processor	Little endian processor
alpha	0x01234567	0x67452301
bravo	0x89ab	0xab89
charlie	0xcd	0xcd
delta	0xef	0xef

Variables stored in memory in little endian byte address order (hex representation of byte value is still left-right big endian):

Variable:	alpha	bravo	charlie	delta
Address:	0 1 2 3	4 5	6	7
Byte value:	67 45 23 01	ab 89	cd	ef

Values (left-right big endian) of variables that each processor would interpret if reading from memory where the variables are stored in little endian byte address order:

	Little endian processor	Big endian processor
alpha	0x01234567	0x67452301
bravo	0x89ab	0xab89
charlie	0xcd	0xcd
delta	0xef	0xef

If you want to share data across different byte endian platforms, you need to do the byte swapping in your own program. What the compiler can do for you is generate the necessary byte swapping instructions, if you tell it the byte endian of your data structures. The compiler can generate the right instructions because it knows the size of every variable. But the swap must be performed by program instructions operating on the right size data structure. This can't be done by any interface.

Actually, of course, it can - subject to the requirements listed above. So maybe a particular bus only carries one size and endian of data, between one data producer and one data consumer of known endian. But here's the deal: The data processing horsepower you would save by byte swapping "on-the-fly" is not worth the software maintenance headache you would create. If you change the byte endian of the data structures on the wires, then you can't declare the data structures with the same header file for the code compiled on both producer and consumer. What you would have in the end would be a time bomb. In the general case of variable size and variable endian data on a general purpose interconnect, it can't be done; and in the few limited situations where it's actually possible, it shouldn't be done anyway.

Never byte swap across interfaces! Always preserve byte address order! If it means sacrificing byte significance, so be it. Interfaces don't know byte significance anyway.

There is no such thing as "big endian memory" or "little endian memory". Memory is just linear storage accessed by the byte address coordinate system. It does not govern or affect significance of the stored data. So I was very careful to specify the byte address endian of the variables stored in memory, not the endian of the memory itself.

An address is a number, and a number (the integer quantity) doesn't have endianness (P1). But if an address is represented by a sequence of digits with position dependent significance, such as "address 834" encoded in a program instruction, the digit representation does have endianness. The endian of a digit sequence representing an address is completely independent of the endian of any digit sequence representing the data value stored at that address. A program may need to byte swap a data value fetched from memory, but never the address value it used for the fetch: It's meaningless to byte swap the coordinate system. But what if a program running on one byte address endian processor reads a pointer variable stored in memory by an opposite byte address endian processor? Then the variable (which happens to represent an address) does need to be byte swapped: Before it can be used as an address, it needs to be fetched as a data value of a variable, and to be interpreted as the correct numeric value it must be byte swapped. But then when used as an address to go back to memory to access the different variable that the pointer points to, the address number is not byte swapped back to the data endian it had before, because it's not being used as a data value to be stored in memory, it's being used as an address coordinate value to access memory. However: The second data value fetched from that address may need to be byte swapped; and if the pointer value were modified (e.g. incremented after being used) and then stored back in memory over the original pointer value, then it would need to be byte swapped back to the original pointer variable data endian! Really, it's all perfectly straightforward...

Never byte swap across interfaces! Always preserve byte address order! If it means sacrificing byte significance, so be it. Interfaces don't know byte significance anyway.

Since interfaces don't even know byte significance, there's very little physical meaning to "bus byte endian". But you see it all the time. All it really means is this: The bus byte endian coordinate system is the byte address across the width of the bus, with the lowest address as the origin. If the bus width is multiple bytes, and if increasing byte address across the bus correlates with increasing bit number, then the "bus byte endian" is the same as the bit endian. Could be big, could be little, but they're the same. If increasing byte address correlates with decreasing bit number, then "bus byte endian" is the opposite of the bit endian. Or to put it another way: pretend the bit endian significance order extends across all bytes, and you get a "pretend most significant" byte and a "pretend least significant" byte. Then the bus byte endian is defined by the pretend byte significance and the byte address coordinate system. A single-byte bus has no byte endian at all: It can transport only a single byte value at any time, and a single digit doesn't have endianness (P4). But the data going by can have endianness relative to previous or subsequent data: Different coordinate system (time vs width).

Byte significance is not always conserved. It's conserved across devices with the same byte endian, regardless of bit endian mismatch; it's reversed across devices with opposite byte endian, regardless of bit endian match.

Bus byte endian has **absolutely nothing** to do with the actual byte endian of data present on the bus. All buses, all data transport mechanisms in general, are byte endian neutral; they know bit significance and byte address and nothing more. So what does "network byte order" mean? It means the endian (in the byte address coordinate system) of the supplemental information that the network fabric uses to request and deliver the payload — packet headers, routing addresses, payload size. It does not apply to the payload. The payload is simply a bag of bytes in address order.

EXAMPLES OF BIT SIGNIFICANCE VS BYTE ADDRESS

Consider x86, PowerPC (PPC), and Motorola/Freescale 68K/ColdFire bit and byte endian (in bit number and byte address coordinates):

x86: Bits Little (bit 0 is least significant); Bytes Little (byte 0 is least significant)

PPC: Bits Big (bit 0 is most significant); Bytes Big (byte 0 most significant)

68K: Bits Little (bit 0 is least significant); Bytes Big (byte 0 is most significant)

And just for completeness' sake, let's add the last possible combination to the mix: A hypothetical device that numbers bits big endian, and bytes little endian. (I'm not aware of any device that uses this scheme, but it's just as valid as all the others.)

For processors with 32bit bus interfaces, the correct interconnect of all four processor types on a 32bit bus is:

Byte Address	Bit Number			
	Hyp	x86	PPC	68K
0	24 : 31 (L)	7 : 0 (L)	(M) 0 : 7	(M) 31 : 24
1	16 : 23	15 : 8	8 : 15	23 : 16
2	8 : 15	23 : 16	16 : 23	15 : 8
3	(M) 0 : 7	(M) 31 : 24	24 : 31 (L)	7 : 0 (L)
Hyp bit 24 connects to x86 bit 7, PPC bit 0, and 68K bit 31.				

And for processors with 64bit bus interfaces, the correct interconnect of all four processor types on a 64bit bus is:

Byte Address	Bit Number			
	Hyp	x86	PPC	68K
0	56 : 63 (L)	7 : 0 (L)	(M) 0 : 7	(M) 63 : 56
1	48 : 55	15 : 8	8 : 15	55 : 48
2	40 : 47	23 : 16	16 : 23	47 : 40
3	32 : 39	31 : 24	24 : 31	32 : 39
4	24 : 31	39 : 32	32 : 39	31 : 24
5	16 : 23	47 : 40	40 : 47	23 : 16
6	8 : 15	55 : 48	48 : 55	15 : 8
7	(M) 0 : 7	(M) 63 : 56	56 : 63 (L)	7 : 0 (L)
Hyp bit 56 connects to x86 bit 7, PPC bit 0, and 68K bit 63.				

Each processor's native significance direction, as indicated by "M" (most) and "L" (least), applies to both bit and byte, and is how a numeric value the width of the bus would be presented at the pins of each processor device.

A bus connecting these four devices could adopt any of the four numbering schemes. These are all possible combinations of bit vs byte endian. Bus bit significance is strictly only valid within individual bytes; bus byte significance is simply the extrapolation of the intra-byte bit significance order across the entire bus width, and is completely unrelated to the actual byte endian of any data transferred between any source and destination devices.

- In all cases, the most significant bit number of each byte is shown on the left — left-right big endian, the familiar written endian for numeric values. This

left-right endian is completely independent of the bit number endian. Both endians exist simultaneously, because they're different coordinate systems.

- In all cases, bit significance within the same byte address is preserved. Opposite bit endian components are connected in anti-parallel bit number order, to preserve significance.
- In all cases, the byte address of each 8 bit group is preserved.
- Byte significance is not always conserved. It's conserved across devices with the same byte endian, regardless of bit endian mismatch; it's reversed across devices with opposite byte endian, regardless of bit endian match.

Interfaces haven't the faintest clue what all the data structures mean that fly back and forth; interfaces are responsible only for delivering the data intact. Since interfaces don't know the byte significance or byte endian of the data, the only way to deliver it intact is to **preserve the byte coordinate system.** This is why bytes only have one coordinate system: **Address**. If the byte coordinate system doesn't change, then the byte endian and the byte significance don't change. Byte little endian data fetched by a PowerPC or a 68K processor is still byte little endian; it's still "intact". It's not directly usable by the receiver, because the receiver is a context expecting the endian hardwired into its circuits. When a processor performs an increment operation, the carry propagation logic is wired for a specific significance direction. But it can be made usable just by byte swapping by the program that fetched it. If interfaces along the way did their own byte swapping (just trying to be helpful, of course), the program would need to know the exact path the data took, and what every interface did to it along the way.

The only entity in any system that knows what your data structure means is your own program. Even other programs, even the operating system, have no idea. There are many ways you could construct a portable data structure; the following is not an exhaustive list:

- Choose which byte endian you want the structure to be, and stick with it. On opposite-endian hardware, you'll have to make the program byte swap variables as it reads them in and byte swap them back when it writes them out. This could heavily penalize the opposite-endian hardware, but may be benign on bi-endian hardware.
- Declare one variable in the data structure whose value indicates the endianness of the rest of the data structure. Use a byte variable, which doesn't need to be swapped across platforms. If you transfer a data structure from an opposite-endian machine, the program running on the new hardware has to byte swap the data structure (knowing the sizes and boundaries of all variables) and change the endianness byte, and from then on can operate in its hardware native endianness.
- Let the program work with the data in the preferred endianness of whatever hardware it's running on, but always convert from and to a consistent endian when loading and saving files.
- Let the program work with the data in the preferred endianness of whatever hardware it's running on, but always convert from and to a file format that does not need to be byte swapped at all: a file of bytes, such as an ASCII "text" file. Big endian and little endian machines alike both store strings in big endian form: first (most significant) character at the lowest address.

Finally, we must consider bi-endian hardware. Static bi-endian hardware can be configured to adopt either byte endian as the "apparent native" form, and bytes are automatically swapped in hardware from the "apparent native" form to the true internal native form. Static bi-endian hardware requires all data in the current "apparent native" byte endian form; opposite byte endian data still needs to be byte swapped in software. (In static bi-endian hardware, if the true native endian is the opposite of the apparent native endian, the hardware swap is performed on all data, so data that's actually in the true native form to begin with must be swapped by software to undo the hardware swap.) Dynamic bi-endian hardware can operate on individual variables in either endian form. Dynamic bi-endian hardware needs some way to know the endianness of any particular piece of data. One possibility is to specify endianness on a page-by-page basis. When reading or writing one page endianness, no byte swapping is done. When reading or writing the opposite page endianness, bytes are automatically swapped in hardware.

The only entity in any system that knows what your data structure means is your own program. Even other programs, even the operating system, have no idea.

If you design hardware, it is your job to preserve bit significance and byte address throughout your system and everywhere your system interfaces with other systems.

Dynamic or static, bi-endian hardware works because the hardware swap is performed internal to the processor, triggered by instructions that reference memory operands, and is performed **only on the data size specified by the instruction.** Assuming you fetch a 4-byte variable with an instruction that fetches 4 bytes, the processor hardware will swap exactly 4 bytes prior to operating on the data. No matter the width(s) of any interface(s) the data crossed to get there. But don't use a single 4-byte instruction to fetch two 2-byte variables! (You shouldn't need to worry about it in compiled code. The compiler knows the size of every variable, and should generate the right size instruction for any variable you reference.)

As you've probably guessed by now, you impose a context on a region of memory by giving it to a program that expects a certain format. The program instructions will interpret the data according to the context of the hardware construction (such as the wiring of carry propagation), regardless whether the data conforms to those expectations. Context does not determine endianness.

If you design hardware, it is your job to preserve bit significance and byte address throughout your system and everywhere your system interfaces with other systems.

If you design software, it is your job to byte swap as necessary based on the byte endianness of your data structures and the byte endianness of the machine you're running on. If at all possible, delegate to the compiler. You should never need to bit swap (unless you're coding algorithms that employ bit reversal, like FFT).

If a processor has cache, the data will not be byte swapped in the cache. Instead, it will be byte swapped between the first-level cache and the internal register file, or between the cache and the ALU for a processor that can perform operations directly on memory operands. All levels of cache are just fast aliases for regions of memory, and data in cache still has memory addresses, and byte address must be conserved across the entire memory hierarchy. The register file and the ALU do not have memory address; the data is swapped according to the current setting of apparent native endian vs true native endian whenever it's removed from or returned to the memory address coordinate system.

Returning to our fictitious XYZ Corp's camera phone chip, once they understood the error, the fix was trivial: Simply declare the camera data little endian in a C header file and recompile the embedded code. Okay, maybe they got lucky, but really they did almost everything right.

If you ever need to deal with endianness, just remember these four words:

Bit significance, Byte address.

And pay attention to coordinate systems!

ABOUT THE AUTHOR:

Kevin has over 25 years experience in the design and verification of digital ASIC's, processor cores and SoC's. Since joining Verilab in January of 2006, Kevin has assisted with the verification of a multi-threaded DSP core and taught a graduate course in processor architecture at the Universidade Federal do Rio Grande do Sul, Porto Alegre in Brazil. Prior to Verilab, Kevin worked at Motorola/Freescale designing and verifying 68K- and PowerPC-based SoC's, and at Texas Instruments as a design and test engineer.